Recognizing Digits from Natural Handwriting Using ML

Introduction

Handwriting has long been an essential part of communication in society but is quickly declining in popularity as alternative non-verbal means of communication are further enabled through technology. However, particularly during this transition phase between handwritten and digital means of communication, the ability for computers to recognize natural handwriting can significantly assist in automating repetitive and mundane tasks. As an example, this technology could be paired with robots to sort packages into appropriate destination hubs (as opposed to a human reading labels and sorting packages manually). Moreover, as longstanding businesses continually transition to digital systems, the need to migrate large amounts of physical documents (printed text) still push for continuous development of OCR (optical character recognition) to automate the process of digitizing critical documents. This however was a simpler problem as printed characters are consistent and can be referenced from common fonts. Natural handwriting is much more complex as characters are dissimilar and unique to each writer - yet handwritten text is consistently understandable to the human reader.[MLBook]

Handwriting recognition is also becoming increasingly relevant in the tablet industry which is bridging the gap between handwriting on paper and integrating complex features to digital handwriting. Such features allow users to edit and interact with handwritten text in entirely novel ways [DilmeganiCem]. Being a user of such technology myself, I find the technology enabling the existence of such features to be fascinating.

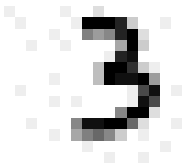
Problem Formulation

I will focus on recognizing the handwritten digits 0 to 9 to limit the scope of this application. This can be modelled as a machine learning problem with data points representing individual digits. Each digit can be characterized by the light and dark pixels on a grid of a set resolution, hence the features (input data) of our problem. This input data can be extracted from downscaled photos of hand drawn digits – however, to eliminate the problem of image noise from lighting/paper/camera, we will use a stylus and drawing tablet to capture the hand drawn digit digitally. Our labels (quantity of interest) on each datapoint are the respective classes of each character, or in our case the corresponding digit to the input character (hand drawn digit). More specifically features could include histograms counting the number of black pixels along vertical and horizontal directions and the number of internal holes or enclosed spaces.

An alternative source of training data can be found from sources such as the EMNIST dataset which is a large dataset with nearly a million handwritten digits. It is a subset of an even larger NIST dataset containing all characters. [EMNIST]

Finally, this is a classification problem, as we need ML to recognize what group the character belongs in, or simply to classify it. There is no middle ground between options, as every character must fall into "0,1,2,3,4,5,6,7,8,9". This is also why this is impossible to model as a regression problem. Consequently, these are sorted into a dataset with each data point labeled with the corresponding digit, and an additional dataset with unlabelled data points which can be used to assess the quality of the model we will eventually select.[MLBook]

Method



I created my dataset by handwriting 10 versions of every digit, which I collected into a labeled repository of .PNG files. These were then scaled down to a 8x16 pixel resolution to eliminate unnecessary features as large resolutions add pixels which bring no additional value for processing and slow down the algorithm. The smallest resolution under which a character is recognizable would be 3x5 (the needed resolution to draw the smallest character on a LED matrix display) but this would not be applicable to handwriting as the characters would be unrecognizable as soon as they are even slightly different from the reference value.

Two appropriate methods to solve this ML problem would be the k-NN algorithm and decision trees. The k-NN (k- nearest neighbours) algorithm is a supervised ML algorithm which relies on labeled input data to learn a function which would produce the appropriate output when we eventually present it with the unlabelled dataset. As a classification problem, we will get a discrete value as an output as opposed to a regression problem. The KNN algorithm assumes that similar things exist in close proximity to one another: for each example in the data, the distance between the query example and ordered collection is calculated and added to an ordered collection of distances from smallest to largest, then the first K entries are picked from the sorted collection, and the mode of the K labels is calculated to give a corresponding discrete output value [EDUCBA]. When using k-NN we also need to specify a K value, which determines the number of nearest neighbours - in practical terms this is the number of categories which in our case is 10. The foundational hypothesis space on which k-NN problems are constructed consists of all maps $h : X \rightarrow Y$ where the value of $h(x)$for a particular feature vector x depends only on the k nearest data points of some training data [EDUCBA]. This can easily be implemented in python using SciKitLearn libraries.

An alternative model to use would be decision trees, a white box type of ML algorithm (no hidden layer). [CallMiner] This type of ML algorithm is faster to train compared to complex ML methods such as neural networks, but functions by selecting the best attributes to split the records, then breaking that attribute up into smaller subsets, and finally repeating this process recursively for each child until one of the conditions matches in order to produce output [DC DT]. This flowchart-like method which maps $h : X \rightarrow Y$of features $x \in X$of data points to a predicted label $h(x) \in Y$can be used for arbitrary feature space $X$and label space Y[MLBook]. The decision tree works through a network of connected nodes, through which it finds step-by-step instructions to link new input to the correct label.

Results

We can evaluate the success of these methods using loss functions such as MSE (mean-squared-error) to calculate the accuracy of our methods/algorithms.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(e_i)^2 = \frac{1}{n}e^T e$$

I implement this using SciKitLearn in python to calculate my the training and validation errors after training the k-NN and decision tree models to get the following results:

	Training error	Validation error
k-NN mean squared error:	≈4.65628 %	≈6.35 %
Decision tree mean squared error:	≈11.35425 %	≈17.28 %

According to the results from our calculations in python, the k-NN model was more consistent at estimating the correct labels for given input (performed better). While earlier a 80/20 split between my data for training and validation respectively was used, I tested this with some new data points which I generated (manually created) and the accuracy was consistent with the calculated accuracies.

Conclusion

The KNN has advantages in its simplicity and ease of implementation, it also has no need to build a model, tune parameters besides K, or make additional assumptions [EDUCBA]. The algorithm is also versatile, and can be used for classification, regression, and searches, but the drawback is in the scalability of the algorithm as it will get significantly slower as the number of examples or predictors and independent variables increase. This disadvantage of becoming slower as the volume of data increases makes it an impractical choice when expanding the character sets to the hundreds of characters from combined capital and lowercase letters along with symbols.

My results suggest that the k-NN algorithm is clearly more successful at recognizing handwriting or categorizing the images, but this could also be a consequence of the input data which is based solely on my handwriting and might have introduced a bias in the training. 100 data points is also a very limited training sample space, and expanding on this at higher resolutions is guaranteed to produce much more accurate results. Overall k-NN algorithm proved to be the method of choice between decision trees and k-NN for character recognition in our experiment. However, it is known that neither algorithm is ideal for high dimension ML applications, hence increasing resolution and data points on our training data might prove very inefficient or demanding. As a conclusion it would be worthwhile looking into other machine learning algorithms for character recognition especially when moving towards the larger field of general character recognition (includes the majority of ASCII characters).

References

[MLBook] A. Jung, “Machine Learning. The Basics”, 2021, mlbook.cs.aalto.fi

[DilmeganiCem] Cem DilmeganiCem founded AIMultiple in 2017. Throughout his career, et al. “Handwriting Recognition in 2021: In-Depth Guide.” AIMultiple, 1 Jan. 2021, research.aimultiple.com/handwriting-recognition

[EMNIST] https://www.nist.gov/itl/products-and-services/emnist-dataset

[DataCamp] “Decision Tree Classification in Python.” DataCamp Community, www.datacamp.com/community/tutorials/decision-tree-classification-python.

[CallMiner] CallMiner. “Machine Learning Algorithms: A Tour of ML Algorithms & Applications.” CallMiner, 18 June 2020

[EDUCBA] “KNN Algorithm: Steps to Implement KNN Algorithm in Python.” EDUCBA, 3 Mar. 2021, www.educba.com/knn-algorithm/.

[DC DT]“Decision Tree Classification in Python.” DataCamp Community, www.datacamp.com/community/tutorials/decision-tree-classification-python.

Code Appendices:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import  train_test_split
from sklearn.metrics import classification_report
from pyimagesearch import simplepreprocessor
from pyimagesearch import simpledatasetloader
from imutils import paths
import os
import glob
import cv2
import numpy as np

def getListOfFiles(dirName):
    # create a list of file and sub directories
    # names in the given directory
    listOfFile = os.listdir(dirName)
    allFiles = list()
    # Iterate over all the entries
    for entry in listOfFile:
        # Create full path
        fullPath = os.path.join(dirName, entry)
        # If entry is a directory then get the list of files in this
directory
        if os.path.isdir(fullPath):
            allFiles = allFiles + getListOfFiles(fullPath)
        else:
            allFiles.append(fullPath)
```

```
    return allFiles

imagePaths = getListOfFiles("./datasets/") ## Folder structure:
datasets --> sub-folders with labels name
#print(imagePaths)

data = []
lables = []
c = 0 ## to see the progress
for image in imagePaths:

    lable = os.path.split(os.path.split(image)[0])[1]
    lables.append(lable)

    img = cv2.imread(image)
    img = cv2.resize(img, (32, 32), interpolation = cv2.INTER_AREA)
    data.append(img)
    c=c+1
    print(c)

#print(lables)

# encode the labels as integer
data = np.array(data)
lables = np.array(lables)

le = LabelEncoder()
lables = le.fit_transform(lables)

myset = set(lables)
print(myset)

dataset_size = data.shape[0]
data = data.reshape(dataset_size,-1)

print(data.shape)
print(lables.shape)
print(dataset_size)

(trainX, testX, trainY, testY ) = train_test_split(data, lables,
test_size= 0.25, random_state=42)

model = KNeighborsClassifier(n_neighbors=3, n_jobs=-1)
model.fit(trainX, trainY)

print(classification_report(testY, model.predict(testX),
target_names=le.classes_))
```